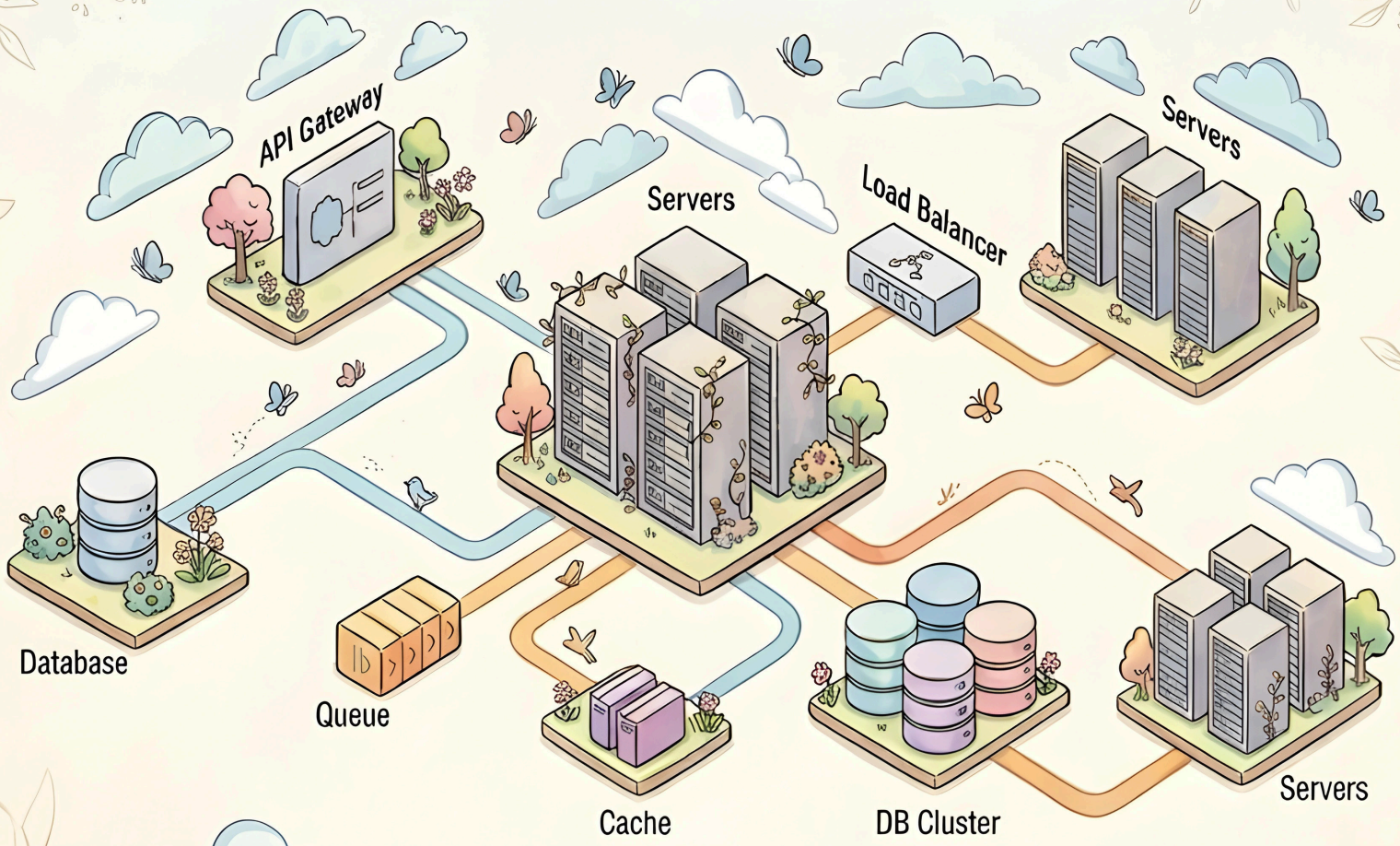
{system design} unboxed

A Visual Reference



LeXTrem Books

This is a limited preview of the book.

You can get the complete edition, including all **17 chapters**, the **PDF** and **EPUB** versions, and the editable diagram source files in **Mermaid**, **SVG**, and **draw.io**, by visiting:

books.lextrem.com/system-design-1

{system design}

unboxed

A Visual Reference

System Design Unboxed: A Visual Reference

Volume 1: Foundations, Storage & Communication Patterns

by Rick Sanrose

Copyright © 2026 LexTrem Books. All rights reserved.

No portion of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means — electronic, mechanical, photocopy, recording, scanning, or otherwise — except for brief quotations in critical reviews or articles, without the prior written permission of the publisher.

Publisher, Cover Design, Illustrations: LexTrem Books

Edition/version 1.0 (August 2026)

ISBN: [ISBN yet to be assigned] (eBook)

Disclaimer: This book is presented for educational and informational purposes only. The authors and publisher assume no responsibility for errors, omissions, or damages resulting from the use of the information contained herein. All product names, logos, and brands are property of their respective owners.

Fonts and Icons: The body text is set in Bitter by Sol Matas, licensed under the SIL Open Font License. Code blocks use Google Sans Code by Google, licensed under the Apache License 2.0. Icons are from Phosphor by Tobias Dahm and contributors, licensed under the MIT License.

TABLE OF CONTENTS

- DEDICATION
- COLOPHON
- ABOUT THIS BOOK
- PROLOGUE: WHY SYSTEM DESIGN (AND WHY NOW)
- ACRONYMS QUICK REFERENCE
- How to Read This Book
- PART I: THE LANDSCAPE OF DISTRIBUTED SYSTEMS
  - CHAPTER 1: FOUNDATIONS OF DISTRIBUTED SYSTEMS
    - 1.1 The Trouble with Distributed Systems
    - 1.2 Scalability: Vertical vs. Horizontal
    - 1.3 The CAP Theorem and Why It Rules Your Life
    - 1.4 Consistency Models: A Hierarchy of Pain
    - 1.5 Load Balancing: The Traffic Cop
    - 1.6 Caching: The Universal Performance Solution
    - 1.7 Data Partitioning (Sharding)
    - 1.8 Proxies: Forward, Reverse, and Everything In Between
    - 1.9 Redundancy and Replication
    - 1.10 Quick Reference
  - CHAPTER 2: HOW TO SOLVE ANY DESIGN PROBLEM
    - 2.1 The Four-Step Design Framework
    - 2.2 The Meta-Skill: Articulating Trade-offs
    - 2.3 The Toolbox Map
    - 2.4 Quick Reference
  - CHAPTER 3: BACK-OF-THE-ENVELOPE ESTIMATION
    - 3.1 The Five-Step Estimation Framework
    - 3.2 Reference Numbers
    - 3.3 Worked Example: Twitter-Scale Feed
    - 3.4 Common Pitfalls
    - 3.5 Quick Reference
- PART II: THE ALGORITHMIC TOOLBOX
  - CHAPTER 4: CONSISTENT HASHING
    - 4.1 Requirements and Data Model
    - 4.2 Architecture Detail

        - 4.3 Algorithm Comparison
        - 4.4 Evolution Path
        - 4.5 Quick Reference
    - CHAPTER 5: RATE LIMITER
        - 5.1 Requirements and Data Model
        - 5.2 Architecture Detail
        - 5.3 Key Algorithms and Decision Matrix
        - 5.4 Evolution Path
    - CHAPTER 6: UNIQUE ID GENERATOR
        - 6.1 Requirements and Data Model
        - 6.2 Architecture Detail
        - 6.3 Algorithm Comparison
        - 6.4 Evolution Path
        - 6.5 Quick Reference
- PART III: STORAGE AND PERSISTENCE
    - CHAPTER 7: URL SHORTENER
        - 7.1 Requirements and Data Model
        - 7.2 Architecture Detail
        - 7.3 Algorithm Comparison
        - 7.4 Evolution Path
        - 7.5 Quick Reference
    - CHAPTER 8: DISTRIBUTED KEY-VALUE STORE
        - 8.1 Requirements and Data Model
        - 8.2 Architecture Detail
        - 8.3 Algorithm Comparison
        - 8.4 Evolution Path
        - 8.5 Quick Reference
    - CHAPTER 9: DISTRIBUTED FILE STORAGE
        - 9.1 Requirements and Data Model
        - 9.2 Architecture Detail
        - 9.3 Algorithm Comparison
        - 9.4 Evolution Path
        - 9.5 Quick Reference
- PART IV: DATA PROCESSING AT SCALE
    - CHAPTER 10: WEB CRAWLER
        - 10.1 Requirements and Data Model

        - 10.2 Architecture Detail
        - 10.3 Algorithm Comparison
        - 10.4 Evolution Path
        - 10.5 Quick Reference
    - CHAPTER 11: DISTRIBUTED MESSAGE QUEUE
        - 11.1 Requirements and Data Model
        - 11.2 Architecture Detail
        - 11.3 Algorithm Comparison
        - 11.4 Evolution Path
        - 11.5 Quick Reference
- PART V: REAL-TIME COMMUNICATION
    - CHAPTER 12: NOTIFICATION SYSTEM
        - 12.1 Requirements and Data Model
        - 12.2 Architecture Detail
        - 12.3 Algorithm Comparison
        - 12.4 Evolution Path
        - 12.5 Quick Reference
    - CHAPTER 13: NEWS FEED SYSTEM
        - 13.1 Requirements and Data Model
        - 13.2 Architecture Detail
        - 13.3 Algorithm Comparison
        - 13.4 Evolution Path
        - 13.5 Quick Reference
    - CHAPTER 14: CHAT AND MESSAGING SYSTEM
        - 14.1 Requirements and Data Model
        - 14.2 Architecture Detail
        - 14.3 Algorithm Comparison
        - 14.4 Evolution Path
        - 14.5 Quick Reference
- PART VI: MEDIA DELIVERY INFRASTRUCTURE
    - CHAPTER 15: VIDEO STREAMING PLATFORM
        - 15.1 Requirements and Data Model
        - 15.2 Architecture Detail
        - 15.3 Algorithm Comparison
        - 15.4 Evolution Path
        - 15.5 Quick Reference

    - CHAPTER 16: CDN DESIGN
        - 16.1 Requirements and Data Model
        - 16.2 Architecture Detail
        - 16.3 Algorithm Comparison
        - 16.4 Evolution Path
        - 16.5 Quick Reference
- PART VII: OPERATING THE BEAST
    - CHAPTER 17: MONITORING AND OBSERVABILITY
        - 17.1 Requirements and Data Model
        - 17.2 Architecture Detail
        - 17.3 Algorithm Comparison
        - 17.4 Evolution Path
        - 17.5 Quick Reference
- APPENDIX: DECISION MATRICES AND QUICK REFERENCES
    - CAP Trade-offs in Practice
    - Latency Numbers Every Engineer Should Know
    - System Design Reference Numbers
    - Decision Index
    - Canonical Design Thresholds
- GLOSSARY
- BIBLIOGRAPHY
    - Books
    - Papers
    - Additional Books
    - Standards and Specifications
    - Architecture References

SYSTEM DESIGN UNBOXED: A VISUAL REFERENCE
Volume 1
Foundations, Storage & Communication Patterns

DEDICATION

To every developer who has ever said "it works on my machine" and then watched it burn in production. To every engineer who has been asked to "just add one more microservice" at 4 PM on a Friday. To every soul who has stared at a CAP theorem Venn diagram and wondered why anyone would willingly choose this career.

This one is for you. May your caches be hot and your p99s be low.

> *"A distributed system is one where the failure of a computer you did not even know existed can render your own computer unusable."*
>
> — Often attributed to Leslie Lamport; possibly Andrew Tanenbaum. The exact origin is uncertain.

COLOPHON

This book was written, typeset, and illustrated with love (and a moderate amount of profanity). No trees were harmed in the production of this edition, though several coffee plantations were depleted. The sarcasm is organic and free-range.

All source code in this book is pseudocode. It will not compile. It is not meant to. If you try to run it in production, the authors assume no liability for the resulting pager storm.

ABOUT THIS BOOK

Who This Book Is For

You are a software engineer with anywhere from one to eight years of experience. You have built a CRUD application — maybe in your sleep, maybe while still figuring out the best way to structure it. You can argue about tabs versus spaces with religious conviction. But when someone asks you to "design a system that handles 10 million users," your brain produces nothing but the sound of a dial-up modem.

Alternatively, you have been tasked with designing a distributed system at work and have discovered that your company's "architecture" consists of a monolith held together with duct tape and hope.

This book is for you.

And if you are earlier in your career and hungry to learn: welcome. Read it once to get the lay of the land, then come back to it as you gain experience. It will be waiting, and you will get more out of it each time.

Who This Book Is NOT For

- Architects who think PowerPoint is a programming language.
- People who unironically use the word "synergy" in technical discussions.

- Anyone who believes that 600-page academic textbooks are an appropriate way to learn system design.
- Your manager, who will inevitably ask you to present a summary at the next all-hands.

What You Will Learn

By the end of this volume, you will be able to:

- Explain the difference between vertical and horizontal scaling without using the phrase "bigger box."
- Design a rate limiter that does not collapse under its own traffic.
- Describe consistent hashing in a way that does not require a whiteboard and forty-five minutes.
- Architect a notification system that actually delivers notifications (most of the time).
- Understand why your chat application occasionally shows messages out of order and how to fix it.
- Design a distributed system without visibly sweating.

Prerequisites

This book assumes you build or work with real-world applications. You should be comfortable with:

- **Basic networking:** You know what an IP address, a port, TCP, DNS, and HTTP are.
- **Databases:** You have used at least one relational database and understand indexes, queries, and connection pools.
- **Operating systems:** You understand processes, threads, memory, disk I/O, and the difference between sequential and random access.
- **Programming:** You can read pseudocode and reason about concurrency (locks, race conditions, atomic operations). Some professional experience helps, but is not required — the book teaches everything from first principles.
- **Math:** You are comfortable with Big-O notation, basic probability, throughput calculations (QPS — Queries Per Second, daily volume), and capacity estimates (TB/PB conversions). No calculus or advanced statistics required.

No prior distributed systems knowledge is required — that is what this book teaches. If you understand the bullet points above, you have everything you need, regardless of whether you have two years of experience or two months.

What You Will NOT Learn

- How to tune the JVM garbage collector (that is a deep rabbit hole).
- The complete specification of the Raft consensus algorithm (read the paper if you have a weekend to spare).
- How to center a div (some mysteries are best left unsolved).

The Author

Rick Sanrose is an engineer with extensive experience designing and building distributed systems at scale. Over the years, he found himself wishing for a concise, visual reference that made these concepts easier to understand. So he decided to write one.

PROLOGUE: WHY SYSTEM DESIGN (AND WHY NOW)

The Situation

Once upon a time, building a web application was simple. You had a server, a database, and a prayer. If your application became successful enough that you needed more than one server, you bought a bigger server. This was called "scaling up," and it worked beautifully until it did not.

The moment your user base crossed a certain threshold — perhaps 10,000 concurrent users, perhaps a million — the monolith began to show cracks. The database connection pool became a battleground. The cache invalidation strategy (which was, let us be honest, just a hopeful TTL) started producing bugs that looked like Heisenbugs. The deployment process, once a simple `git push`, became a fragile pipeline that everyone feared to touch.

This is the moment when system design stops being an academic exercise and becomes a survival skill.

The Context

We live in an age of unprecedented scale. The largest social media platforms handle billions of daily active users. E-commerce sites process millions of transactions per hour during peak seasons. Video streaming services deliver petabytes of data every single day to every corner of the globe. And the expectations have never been higher: users expect responses in milliseconds, availability of 99.99% or better, and the ability to access their data from any device, anywhere, at any time.

The tools we use to build these systems have evolved accordingly. Containers, orchestrators, service meshes, event-driven architectures, data lakes, stream processors — the landscape of distributed systems has become a jungle of competing technologies, each with its own trade-offs, its own jargon, and its own passionate community of advocates who will argue until they are blue in the face that their chosen tool is the One True Way.

And yet, beneath all of this complexity, the fundamental problems remain remarkably consistent:

- How do we store data reliably across multiple machines?
- How do we ensure that data remains consistent when networks fail?
- How do we distribute load across servers without creating new bottlenecks?
- How do we handle traffic spikes without provisioning for peak capacity?
- How do we know when something is broken, and how do we fix it before users notice?

These are the questions that real-world architecture decisions answer. And these are the questions that this book will help you answer.

The Motivation

Why should you care about system design?

Career advancement. The difference between a senior engineer and a staff engineer is often the ability to think in terms of systems rather than individual components. Designing and operating systems at scale requires not just knowledge, but the ability to communicate that knowledge clearly and confidently under pressure.

Practical necessity. The systems you build will grow in complexity. Understanding the architectural patterns that underpin distributed systems will help you make better decisions, avoid common pitfalls, and debug problems that span multiple services.

Intellectual curiosity. There is something deeply satisfying about understanding how the systems we use every day actually work. How does Google search index the entire internet? How does Netflix stream video to millions of devices simultaneously? How does WhatsApp deliver messages with end-to-end encryption at planetary scale? These are fascinating questions, and the answers reveal elegant solutions to difficult problems.

The Approach of This Book

This book takes a visual, schematic approach to system design. Every major concept is accompanied by a diagram. Every algorithm is presented as clean pseudocode rather than language-specific implementation. Every architecture is broken down into its constituent parts, with clear explanations of why each component exists and how it interacts with the others.

We do not waste time on academic formalism. We do not bury you in unnecessary detail. We focus on the essential patterns that appear again and again in real-world systems, and we explain them in a way that is memorable, entertaining, and — most importantly — useful.

The tone is deliberately irreverent. Distributed systems are hard enough without taking ourselves too seriously. If a joke helps you remember the difference between a token bucket and a leaky bucket, then the joke has served its purpose.

A Note on the Diagrams

The diagrams in this book were initially sketched using Mermaid, a text-based diagram description format. This format was chosen for several reasons:

1. **Precision.** Text-based descriptions force clarity. If you cannot describe a diagram, you cannot draw it.
2. **Maintainability.** The diagrams can be modified, version-controlled, and regenerated with minimal effort.
3. **Accessibility.** The diagram source is included inline, so readers can modify and experiment with the diagrams themselves.

All diagrams use a consistent visual style inspired by forest tones — greens, warm earth colors, and muted pastels. This is not an aesthetic choice — consistent visual theming helps information retention, and it looks nice.

ACRONYMS QUICK REFERENCE

Acronyms used throughout the book, with the chapter where each is first introduced.

Acronym	Meaning	First Mention
ABR	Adaptive Bitrate Streaming	Ch. 15
AP	Availability + Partition Tolerance	Ch. 1
APNs	Apple Push Notification service	Ch. 12

Acronym	Meaning	First Mention
BFS	Breadth-First Search	Ch. 10
CAP	Consistency, Availability, Partition Tolerance	Ch. 1
CDN	Content Delivery Network	Ch. 16
CP	Consistency + Partition Tolerance	Ch. 1
CRDT	Conflict-Free Replicated Data Type	Ch. 14
DASH	Dynamic Adaptive Streaming over HTTP	Ch. 15
DFS	Depth-First Search	Ch. 10
DLQ	Dead Letter Queue	Ch. 11
DNS	Domain Name System	Ch. 1
FCM	Firebase Cloud Messaging	Ch. 12
FIFO	First In, First Out	Ch. 11
HLS	HTTP Live Streaming	Ch. 15
ISR	In-Sync Replicas	Ch. 11
LSM	Log-Structured Merge Tree	Ch. 8
NTP	Network Time Protocol	Ch. 6
P99	99th Percentile Latency	Ch. 1
QPS	Queries Per Second	Ch. 1
QR	Quick Reference	Appx
RED	Rate, Errors, Duration	Ch. 17
RPO	Recovery Point Objective	Ch. 8
RTO	Recovery Time Objective	Ch. 8
SLO	Service Level Objective	Ch. 17
SPOF	Single Point of Failure	Ch. 6
SSTable	Sorted String Table	Ch. 8

Acronym	Meaning	First Mention
USE	Utilization, Saturation, Errors	Ch. 17
WAL	Write-Ahead Log	Ch. 8

How to Read This Book

This book has three types of chapters:

Foundational (Ch.1–4): Core concepts, design method, and estimation skills shared by all systems. Read in order.

Application (Ch.5–16): Specific system designs. Each is self-contained once you know the fundamentals; read in any order.

Operational (Ch.17): Observability and monitoring. Best read last, but referenced throughout.

For each application chapter, the structure is: > Requirements → Data Model → Architecture → Algorithm Comparison → Evolution → Quick Reference

If you already know the system, skip to the Quick Reference for a summary. If you are short on time, read Chapters 1 through 4, and 8 — they cover the most generally applicable patterns.

Every chapter ends with a **"You Try"** section — open-ended design exercises meant to stretch your understanding. There are no answer keys; the goal is to practice the reasoning process. The **Quick Reference** at the end of each chapter also serves as a cheat sheet when revisiting.

This is a limited preview of the book.

You can get the complete edition, including all **17 chapters**, the **PDF** and **EPUB** versions, and the editable diagram source files in **Mermaid**, **SVG**, and **draw.io**, by visiting:

books.lextrem.com/system-design-1

CHAPTER 4: CONSISTENT HASHING

Consistent hashing is one of those ideas that seems obvious in retrospect but was genuinely innovative when it was introduced. Before consistent hashing, distributing data across a cluster meant either using a simple modulo operation (which broke when the cluster size changed) or maintaining a routing table (which was a single point of failure and a scalability bottleneck). Both approaches tied the placement algorithm directly to a fixed cluster topology, so any membership change required a complete recomputation of where data should live. As distributed systems grew larger and clusters became more dynamic, this limitation became increasingly difficult to manage in production. The fundamental issue was that the location of each key depended directly on the current number and arrangement of nodes, making ordinary scaling operations surprisingly disruptive.

A production cache cluster that grows from 10 to 11 nodes must move 91% of its keys — and each miss triggers an expensive backend query. Consistent hashing breaks this coupling. It allows the system to add and remove servers while minimizing the amount of data that must be moved. Instead of remapping nearly every key, only the keys belonging to the affected portion of the keyspace are reassigned. This means that adding capacity does not require the system to reorganize the entire dataset, allowing membership changes to be handled incrementally rather than as a full redistribution event. It is used by virtually every distributed data store that needs to partition data across a changing set of nodes: Amazon DynamoDB DeCandia et al., 2007, Cassandra, Riak, Discord, and many others. It replaces an O(N) redistribution problem with one proportional to 1/N, turning a catastrophic event into a routine operation. This property makes horizontal scaling practical because capacity can be increased incrementally without repeatedly relocating the entire dataset. The same principle also makes node removal more manageable, since only the relevant portion of the keyspace needs to be reassigned to the remaining nodes.

> **Real World — Discord's Migration to Consistent Hashing:** Discord originally stored chat attachments using a simple hash of the attachment ID modulo N. When they added new storage nodes, millions of attachments had to be migrated to new locations. They switched to consistent hashing with ~1,000 virtual nodes per server, reducing data movement during cluster expansions from ~(N-1)/N of all data to ~1/N — a difference that turned a weekend-long migration into a rolling update. The important operational benefit was not simply a different hashing formula, but a much smaller amount of data that had to change ownership when the cluster topology changed.

4.1 Requirements and Data Model

In a cluster of N nodes, adding one node with modulo-N hashing requires moving N/(N+1) of all keys; removing one requires moving (N-1)/N — in either case, virtually all data. For a 10 TB cluster with 100 nodes, that is approximately 9.9 TB of network traffic. Consistent hashing reduces this to approximately 1/N (100 GB for the same cluster). This difference determines whether a scaling event takes hours or days. In practice, reducing data movement also shortens recovery times, lowers infrastructure costs, and minimizes the performance impact experienced by users during cluster maintenance or expansion. It also reduces the amount of data that must be transferred, reloaded, or reconstructed when a node joins or leaves the cluster. As a result, cluster membership can change with substantially less disruption to normal request processing, which is one of the main reasons consistent hashing is so useful in distributed systems.

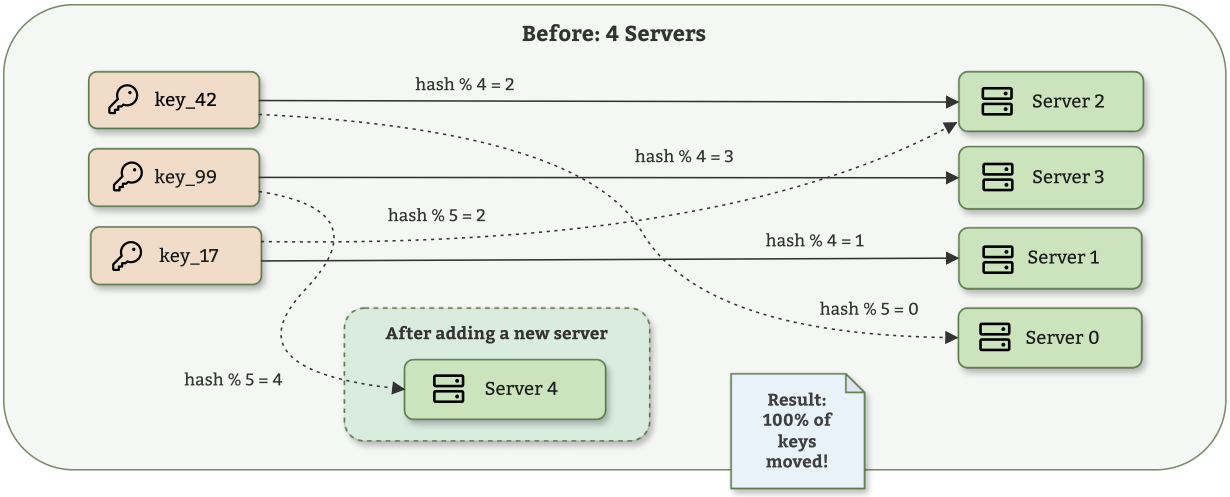


Figure 23: Resharding with Modulo Hashing

To understand why consistent hashing matters, consider what happens when you use a simple modulo operation to distribute keys across servers.

```
With 4 servers:
  hash("user_42") % 4  ->  2  ->  Server 2
  hash("user_99") % 4  ->  3  ->  Server 3
  hash("user_17") % 4  ->  1  ->  Server 1

When you add a 5th server:
  hash("user_42") % 5  ->  0  ->  Server 0   <- moved!
  hash("user_99") % 5  ->  4  ->  Server 4   <- moved!
  hash("user_17") % 5  ->  2  ->  Server 2   <- moved!
```

In this example every key happened to be reassigned, but in general adding a server reassigns on average N/(N+1) of the keys — about 80% when going from 4 to 5 servers. For a cluster with 10 terabytes of data, even moving 80% of it means transferring most of the data across the network. During this migration, the system is vulnerable: if a server fails, the data that was being migrated from it may be lost.

This problem is not limited to adding servers. Removing a server (due to failure or decommissioning) has the same effect. If a server crashes, the modulo calculation changes for every key, requiring a complete redistribution of data.

The solution is to decouple the placement of data from the exact number of servers.

The Ring Architecture

A `SortedMap<uint32, Node>` mapping hash values (0 to 2^{32} - 1) to server nodes. Each key is hashed to a position on the ring; the nearest clockwise server owns it. This structure supports efficient ceiling/floor lookups and dynamic insertion and removal of nodes.

Consistent hashing arranges servers and keys on a conceptual ring. The ring has a fixed range of hash values (typically 0 to 2^{32} - 1). Each server is assigned one or more positions on the ring by hashing its identifier. Each key is also hashed to a position on the ring. In this diagram:

- Key K1 (hash 1200) looks clockwise and finds Server B (hash 2500). It is assigned to Server B.
- Key K2 (hash 3000) looks clockwise and finds Server C (hash 4000). It is assigned to Server C.
- Key K3 (hash 5500) looks clockwise and finds Server D (hash 5800). It is assigned to Server D.

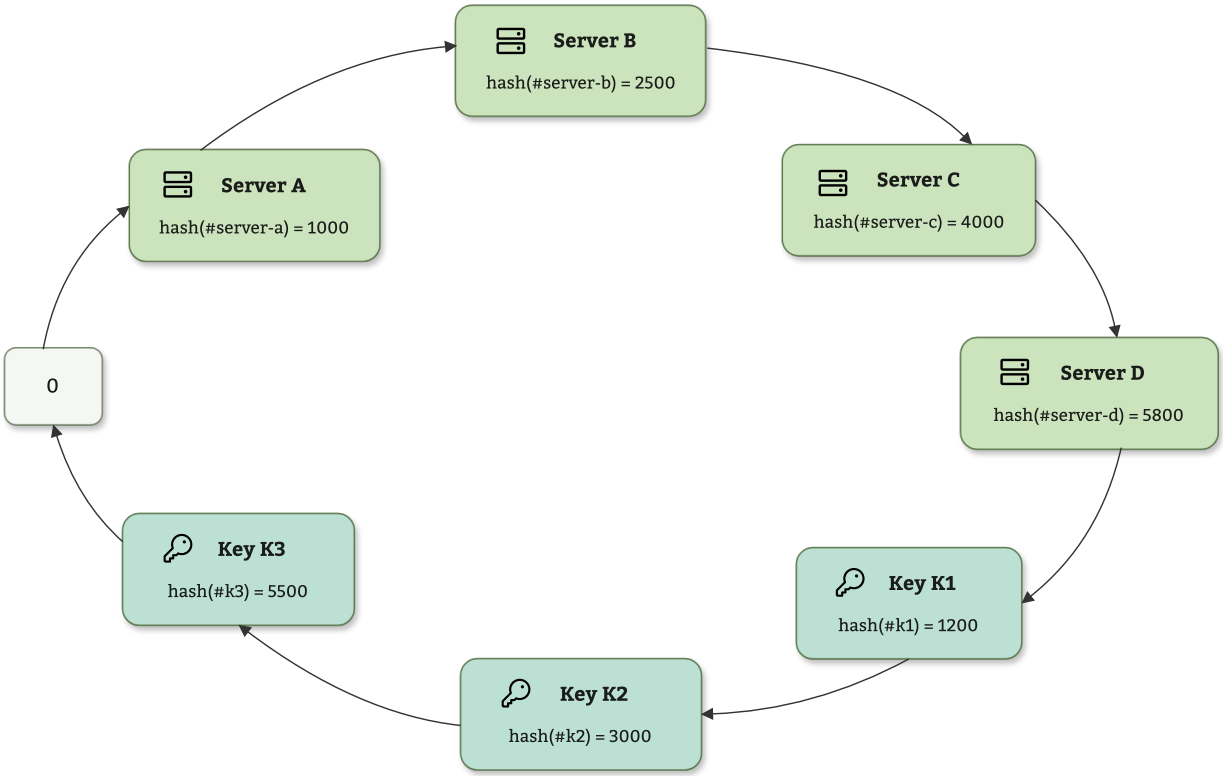


Figure 24: Requirements and Data Model — Consistent Hashing

Now consider what happens when Server B fails. Keys that were assigned to Server B are redistributed to Server C. But keys on Servers A, C, and D remain in place. Only the keys that were on the failed server need to be moved — approximately 1/N of the total keys, where N is the number of servers.

Failure-first: the cascading failure risk. Moving 1/N of keys to Server C sounds manageable — until you consider that Server C is already at 80% capacity. Receiving Server B's traffic pushes it to 120%, causing Server C to fail. Now Server C's keys (including the ones it inherited from B) must be redistributed to Server D, which was already at 75% — and the chain reaction continues around the ring. This is a **cascading failure**, and it is the single most common cause of large-scale consistent hashing outages.

Mitigations:

- **Capacity headroom.** Never run nodes above 60-70% capacity under normal load. The 30-40% headroom absorbs the redistribution spike.
- **Virtual nodes with proportional load.** Distribute the inherited load across *all* remaining nodes (not just the immediate neighbor) by assigning each physical node multiple virtual positions on the ring.
- **Bounded-load consistent hashing.** Reject requests that would exceed a per-node load limit and redirect to the next available node. Google's Slicer Adya et al., 2016 and the consistent hashing with bounded loads paper Mirrokni et al., 2018 formalize this.
- **Gracious degradation under load.** If a node is under stress (latency > p99 threshold), shed 10% of traffic before failing completely — known as **load shedding before failure**.

The difference between standard consistent hashing and bounded-load consistent hashing is straightforward:

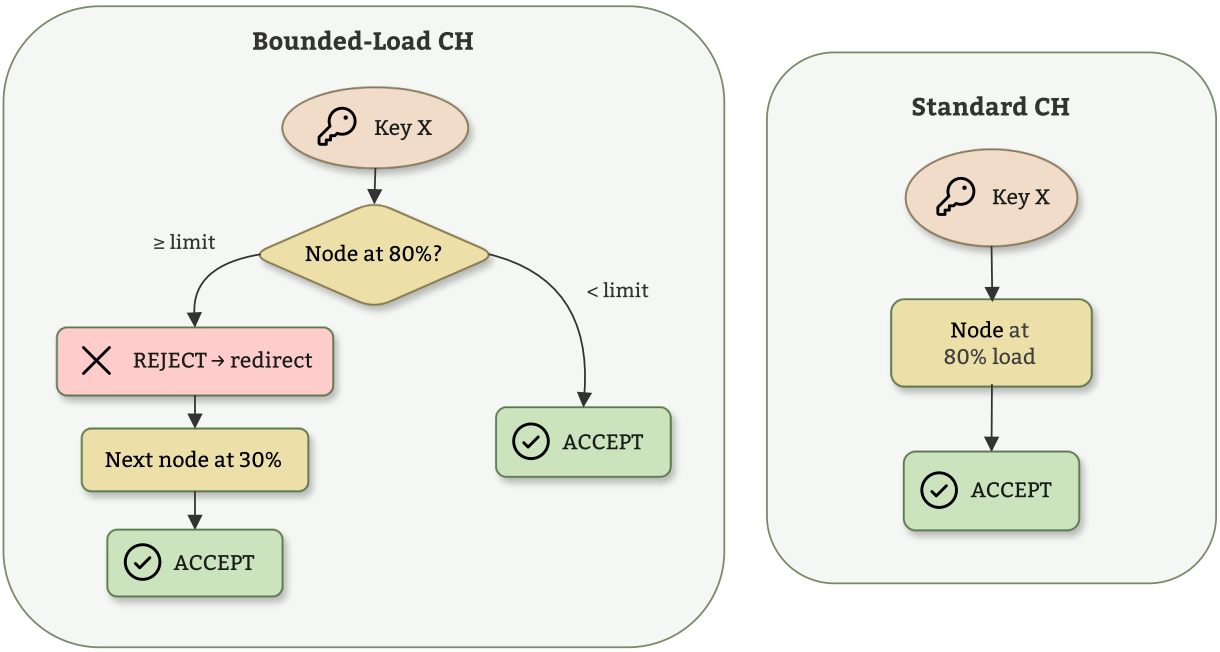


Figure 25: Bounded-Load Hashing

```
With N servers:
  Node failure -> only 1/N of keys are reassigned
  Node addition -> only 1/N of keys are reassigned (to the new node)

Compare with modulo:
  Node failure -> (N-1)/N of keys are reassigned
  Node addition -> N/(N+1) of keys are reassigned
```

This is the key insight of consistent hashing: by organizing servers and keys on a ring and using clockwise assignment, only the keys that were assigned to the failed or added server need to be moved. Everything else stays where it is.

Key Takeaways

- Modulo-N hashing requires moving almost all data when servers are added or removed, making it impractical for dynamic clusters.
- Consistent hashing decouples key distribution from server count, minimizing data movement to ~1/N when resharding.
- Virtual nodes (vnodes) improve load distribution and fault tolerance by assigning multiple positions on the ring per physical server.
- The ring organizes servers and keys in a circular space, assigning each key to the nearest server clockwise.
- *See Section 4.5 Quick Reference for data movement ratios on node changes.*
- The clockwise algorithm enables dynamic scaling without massive migrations.

API Design

```
POST /ring/node
Request:  { "node_id": "server-42", "vnodes": 100 }
Response: { "status": "added", "ring_version": 1042 }
```

```
GET /ring/lookup?key=some_key
Response: { "key": "some_key",
    "assigned_node": "server-7",
    "ring_version": 1042
}
```

```
DELETE /ring/node/{id}
Response: { "status": "removed", "keys_reassigned": 512 }
```

4.2 Architecture Detail

In the basic ring architecture, each server occupies one position on the ring. This creates two problems:

1. **Non-uniform distribution.** With few servers on the ring, the distribution of keys can be highly uneven. If Server A and Server B are close together on the ring, Server A will serve a small range of keys while Server C, which may be far from both, will serve a large range.
2. **Heterogeneous hardware.** Not all servers are created equal. A server with 64 GB of RAM should handle more keys than a server with 16 GB. The basic ring does not account for this.

Virtual nodes (or vnodes) solve both problems. Each physical server is represented by multiple positions on the ring. A server with more capacity gets more vnodes. This spreads the keys more evenly and allows capacity to be allocated proportionally.

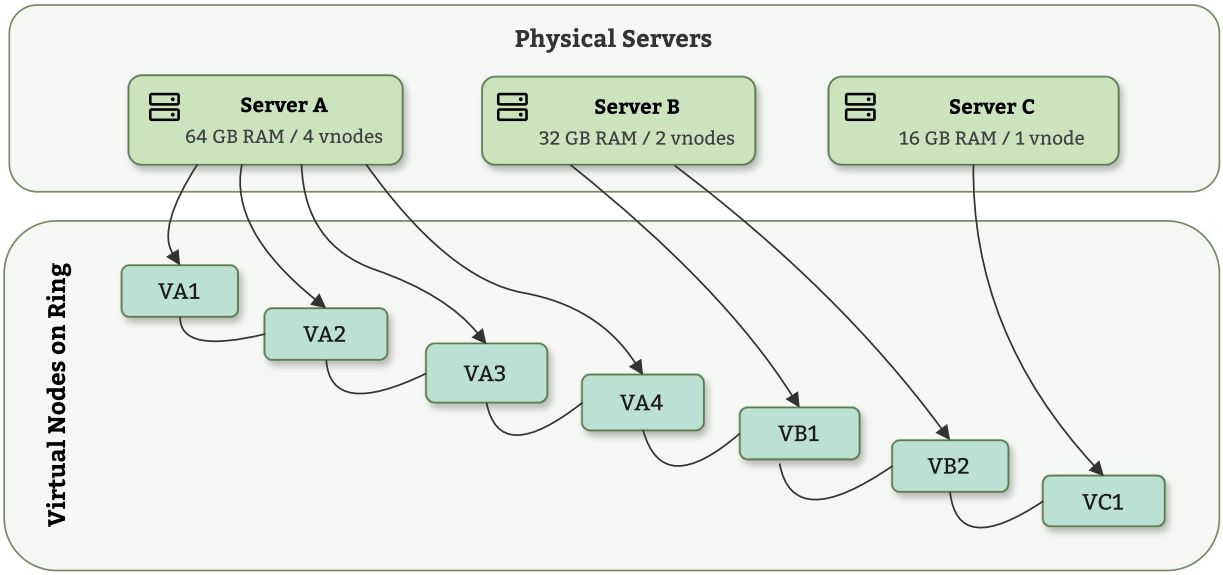


Figure 26: Virtual Nodes

With virtual nodes, the number of positions on the ring is much larger than the number of physical servers. This gives the hash function more opportunities to distribute keys evenly. When a physical server fails, its vnodes are redistributed across the remaining servers, with each remaining server taking a share proportional to its own vnode count.

Virtual nodes also simplify capacity changes. Adding RAM to a server? Increase its vnode count. Decommissioning a server? Remove its vnodes. The ring dynamically adjusts.

Key Takeaways

- Virtual nodes (vnodes) assign multiple ring positions per physical server, improving key distribution evenness.
- Vnodes enable heterogeneous hardware handling by allowing more capable servers to have more positions.
- When a physical server fails, its vnodes redistribute across remaining servers proportionally, maintaining balance.

4.3 Algorithm Comparison

The consistent hash ring is implemented using a sorted map that maps hash values to nodes, with these core operations:

```
CLASS ConsistentHashRing:
    ring: SortedDictionary<uint32, Node>   # hash -> node mapping
    virtualNodeCount: uint32               # vnodes per physical node

    FUNCTION hash(key: String) -> uint32:
        # Note: MD5 is cryptographically broken and slow. Production systems use
        # xxHash (10-20x faster, non-cryptographic) or SHA-256 when needed.
        RETURN xxHash(#key).toUint32()

    FUNCTION addNode(node: Node):
        FOR i IN 1..virtualNodeCount:
            vnodeKey = node.id + ":" + i
            hashValue = hash(vnodeKey)
            ring[hashValue] = node

    FUNCTION removeNode(node: Node):
        FOR i IN 1..virtualNodeCount:
            vnodeKey = node.id + ":" + i
            hashValue = hash(vnodeKey)
            ring.delete(hashValue)

    FUNCTION getNode(key: String) -> Node:
        IF ring.isEmpty():
            THROW Exception("No nodes available")

        keyHash = hash(#key)
        candidate = ring.ceilingEntry(keyHash)

        # Wrap around the ring
        IF candidate IS NULL:
```

```
            candidate = ring.firstEntry()

        RETURN candidate.value

    FUNCTION distributeKeys(keys: List<String>)
        -> Map<Node, List<String>>:
        result: Map<Node, List<String>>
        FOR key IN keys:
            node = getNode(#key)
            result[node].append(#key)
        RETURN result
```

The implementation uses a sorted dictionary (or tree map) to store the ring. The `ceilingEntry` method finds the first entry with a hash value greater than or equal to the key's hash. This is the nearest server clockwise on the ring. If no such entry exists (because the key's hash is larger than all server hashes), the ring wraps around and the first entry is returned.

The `addNode` method creates `virtualNodeCount` positions on the ring for each physical node. Each position uses a different vnode key (constructed by appending an index to the node's ID), ensuring that the hash values are distinct and spread across the ring.

Consistent hashing is used in a wide range of distributed systems:

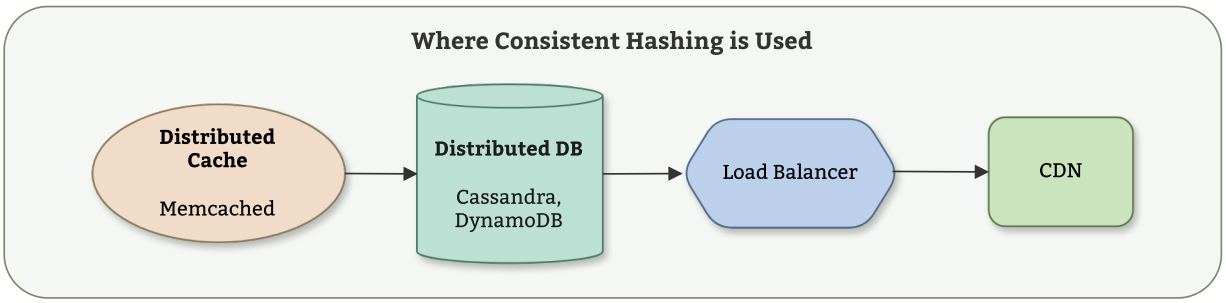


Figure 27: Consistent Hashing Use Cases

Common pitfall: "What happens if a key becomes a hot spot — accessed far more frequently than other keys assigned to the same node?"

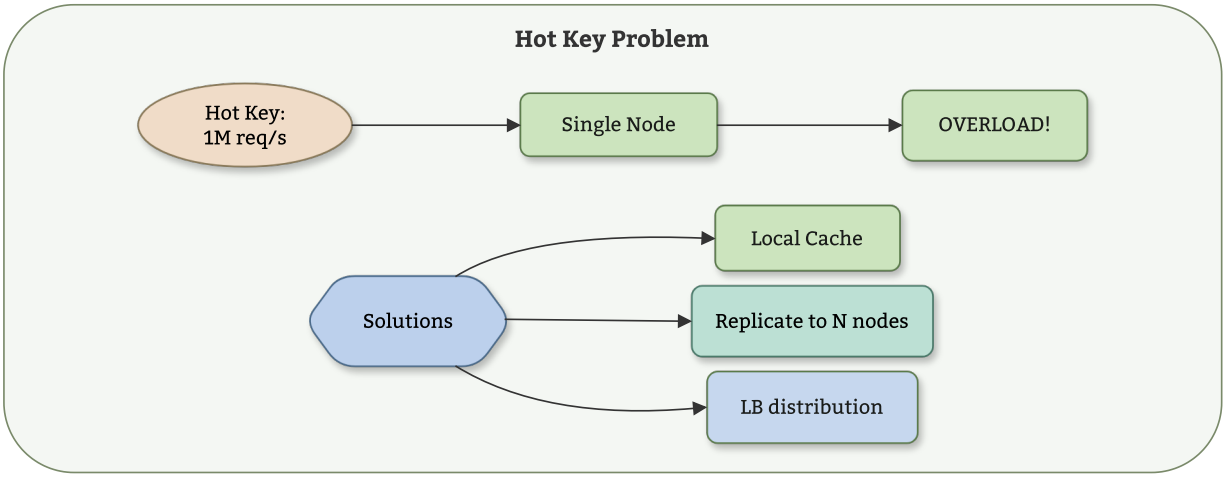


Figure 28: Algorithm Comparison — Hot Key Problem

- **Distributed caches:** Memcached uses consistent hashing to distribute keys across cache servers. Adding or removing a cache server causes minimal disruption.
- **Distributed databases:** Cassandra and Amazon DynamoDB use consistent hashing for data partitioning. Each row is assigned to a node based on its partition key, and virtual nodes ensure even distribution.
- **Load balancers:** Some load balancers use consistent hashing to route requests to backend servers while minimizing disruption when servers are added or removed.
- **Content delivery networks:** CDNs use consistent hashing to assign content to edge servers, ensuring that popular content is cached efficiently.

Decision Matrix

Criterion	Modulo Hashing	Consistent Hashing
Data movement on reshard	Nearly all keys (N-1)/N	Only ~1/N of keys — those whose hash falls in the new node's ring range
Implementation complexity	Trivial: `hash(key) % N`	Moderate: ring, virtual nodes, binary search
Heterogeneous hardware	Not supported	Supported via virtual node weights
Hot key handling	None built-in	Same limitation as modulo
Load distribution	Uniform with good hash	Near-uniform with sufficient vnodes (~100–200 per node)
Coordinator needed	No	No
Production track record	Every hash-based system	DynamoDB, Cassandra, Discord, Akamai

When to choose modulo: Your cluster size is fixed or changes infrequently (cron job maintenance windows). You need the simplest possible implementation and can accept a full data migration during reshard events. Suitable for internal tools with fewer than 10 nodes and planned scaling.

When to choose consistent hashing: Your cluster size changes regularly (auto-scaling, node failures, planned replacements). You need to minimize data movement during scaling events. You operate at a scale where a full reshard takes hours or days. This is the default choice for any production distributed storage system.

Common pitfall: "Consistent hashing eliminates the need for data migration." It does not — it limits the impact to ~1/N of keys instead of (N-1)/N, but migration still happens. Plan for it.

Another pitfall: "Consistent hashing solves hot keys." It distributes keys uniformly, not access frequency. A single popular key can still overload its node.

Solutions include:

- **Local caching:** Cache the hot key's value on the application server to reduce reads to the storage node.

- **Replication:** Replicate hot keys to additional nodes and route reads to any replica.
- **Load balancer override:** For extremely hot keys, route requests through a load balancer that distributes them across multiple nodes.

Key Takeaways

- The ring can be implemented with a sorted map (TreeMap) using ceilingEntry to find the nearest clockwise server.
- Each physical node creates multiple vnode positions with unique hash keys, spreading positions across the ring.
- The algorithm handles ring wraparound by returning the first entry when no larger hash exists.
- **Naive modulo hashing breaks when the cluster size changes**, requiring nearly all data to be moved. Consistent hashing limits the impact to approximately 1/N of keys.
- **Virtual nodes solve both non-uniform distribution and heterogeneous hardware problems** by representing each physical node with multiple positions on the ring.
- **Consistent hashing does not solve hot-key problems.** A single popular key can still overload its node, requiring additional mechanisms like caching or replication.

What Could Go Wrong

Failure	Symptom	Mitigation
Node failure	Keys remapped to other nodes, potential cascading load	Replication + read repair
Uneven load	Hotspot keys overload a single node	Virtual nodes + local caching
Ring stale metadata	Requests routed to wrong node	Gossip protocol for convergence

A **gossip protocol** is a decentralized communication mechanism where each node periodically exchanges state information with a random set of peers. Over time — typically within seconds — every node converges to the same view of the cluster: which peers are alive, what data they hold, and what the ring topology looks like. This works without a central coordinator, making it resilient to node failures and network partitions.

4.4 Evolution Path

Phase 1: Simple Modulo

Phase 1 implements the simplest approach: distributing keys across N servers using hash modulo.

`hash(key) % N`. Works for fixed clusters. Breaks catastrophically when N changes — every key must move. Acceptable only for single-server prototypes or static clusters that never scale.

Phase 2: Ring Without Vnodes

Phase 2 introduces the consistent hash ring, eliminating the reshuffling problem of modulo hashing.

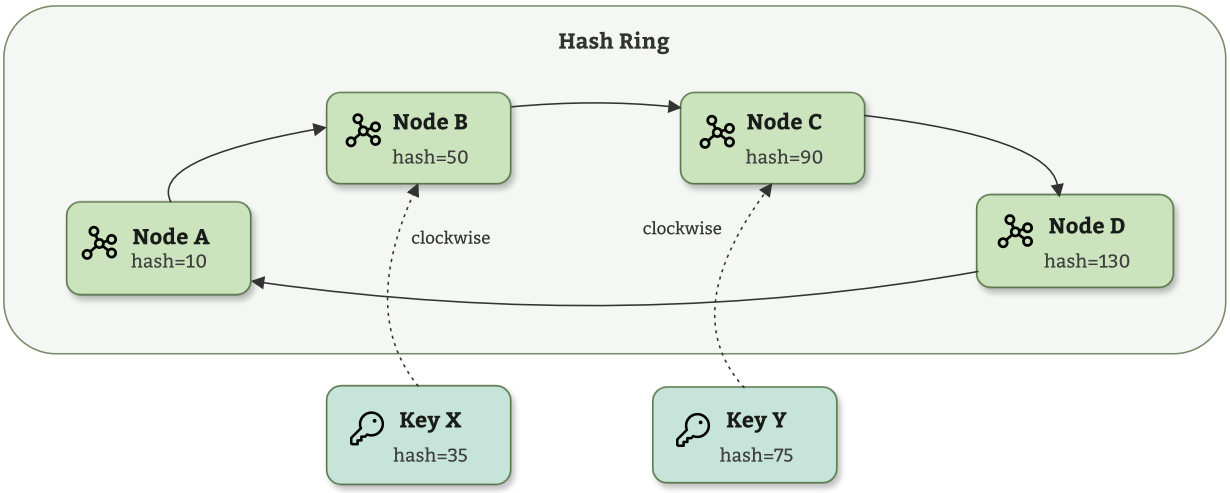


Figure 29: Evolution Path — Hash Ring

The basic consistent hash ring. Adding or removing a node moves only 1/N of keys, a dramatic improvement over modulo. But with few physical nodes, key distribution is uneven — one server may end up with 30% of keys while another has 5%.

Phase 3: Ring With Vnodes

Phase 3 introduces virtual nodes (vnodes), assigning multiple ring positions per physical server for better load distribution.

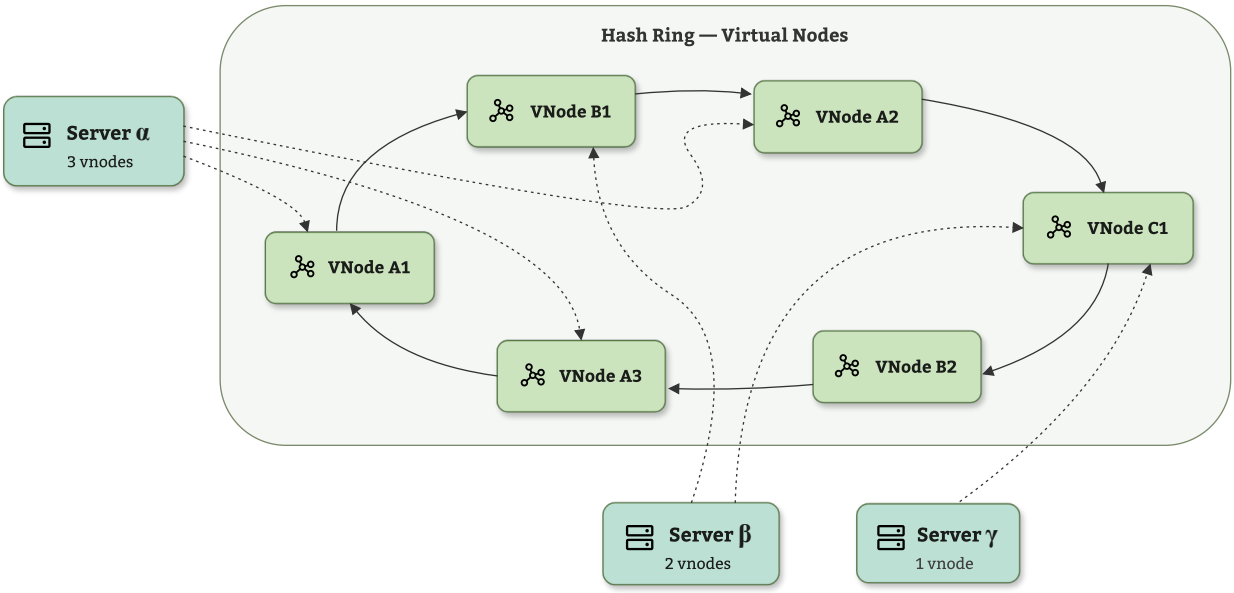


Figure 30: Evolution Path — Virtual Nodes

Each physical server gets 100-200 virtual positions on the ring. Distribution evens out statistically. Heterogeneous hardware gets proportional vnode counts. This is the production standard used by Cassandra, DynamoDB, and Riak.

Phase 4: Replication-Aware Ring

Phase 4 extends the consistent hash ring to be replication-aware, assigning each key to N consecutive nodes on the ring. In production, token assignment is rack/DC-aware (e.g., Cassandra's NetworkTopologyStrategy), so "consecutive on the ring" spans distinct fault domains — the diagram below shows this with nodes across Racks A, B, C.

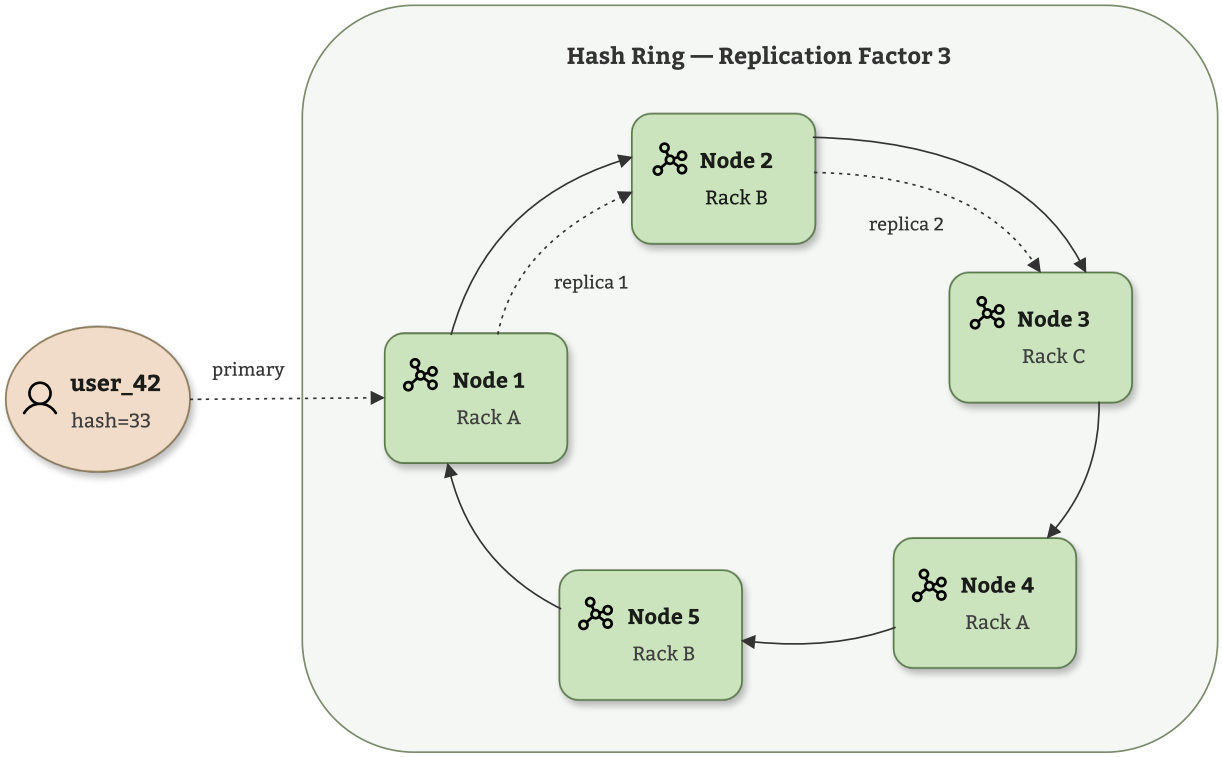


Figure 31: Evolution Path — Replication Factor 3

Each key is assigned to N consecutive nodes on the ring for replication. When a node fails, its replicas handle the load transparently. Rack-aware and datacenter-aware placement ensure fault domain isolation. Used by Cassandra with configurable replication factor.

4.5 Quick Reference

Summary of key concepts for quick reference:

Parameter	Recommendation
Hash function	SHA-256 or xxHash (not MD5 in production)
Virtual nodes per server	100-200 (~1000 for very large clusters)
Ring size	0 to $2^{32} - 1$ (uint32)
Data moved on node change	~1/N of total keys
Hot key mitigation	Local cache, replication, or load balancer override

Decision	Recommendation
When to use	Dynamic clusters, frequent node additions/removals, auto-scaling
When NOT to use	Static clusters (modulo-N is simpler); when hot keys are predictable and dominant
Alternative	Modulo-N hashing for fixed clusters; load balancer with sticky sessions for hot keys

You Try

Three practical exercises to test your understanding:

1. Dynamic node addition without rehashing. Your distributed cache cluster has 32 nodes using consistent hashing with 100 virtual nodes each. The cluster auto-scales to 64 nodes when CPU crosses 70%. A single cache key for a viral video accounts for 15% of all requests. When the cluster scales up, this hot key's virtual nodes migrate to a new physical node, causing that node to instantly hit 90% CPU. Design a "virtual node pinning" strategy that ensures hot keys don't cascade-migrate during scale events. What's the minimum number of virtual nodes needed to keep each node's load within 10% of the mean after a node addition?

2. Discord's migration from modulo to consistent hashing. You work at a chat startup with 50M users. Currently using modulo-N hashing on 10 Redis nodes. You need to add 5 more nodes to handle growth, but modulo-N would invalidate 50% of cached data. Management approves only 2 hours of degraded service. Design a migration strategy: how many nodes can you add per migration batch? What's the optimal intermediate key layout that allows gradual data movement? Use Discord's real-world lesson (100 virtual nodes, ~1/N data moved) as your starting point.

3. Weighted consistent hashing for heterogeneous hardware. Your caching layer consists of 3 node types: 8x large nodes (32 GB RAM, 8 vCPU), 4x medium nodes (16 GB, 4 vCPU), and 20x small nodes (4 GB, 2 vCPU). Consistent hashing with equal virtual nodes gives every node the same key space regardless of capacity. Design a weighted scheme where each node's virtual node count is proportional to its capacity. How does this affect the distribution of keys when a large node fails? Is the failure impact proportional to the node's weight? Show the math.

This is a limited preview of the book.

You can get the complete edition, including all **17 chapters**, the **PDF** and **EPUB** versions, and the editable diagram source files in **Mermaid**, **SVG**, and **draw.io**, by visiting:

books.lextrem.com/system-design-1

System Design Unboxed: A Visual Reference

You can build a CRUD app in your sleep. But when someone asks you to design a system for 10 million users, your brain sounds like a dial-up modem.

System design interviews are the gatekeeper to senior roles. Real-world architecture decisions will determine whether your service survives Black Friday or collapses under its own success. And most resources on the subject fall into two traps: 600-page textbooks you will never finish, or hand-wavy blog posts that skip the hard parts.

This book is neither.

- **Every concept has a diagram**. No walls of text. You see the architecture, the data flow, and the trade-offs at a glance.

- **17 chapters**, from foundations to production. Consistent hashing, rate limiting, distributed storage, CDNs, monitoring — **each pattern explained** with clean pseudocode and real-world rationale.

- **A reference you will actually use**. Read it cover to cover, or jump straight to the chapter you need. The end-of-book TOC, the decision index, and the reference numbers table are designed for quick retrieval when you are in the middle of a design session.

- **Zero fluff**. No academic formalism, no centering divs, no PowerPoint architecture. Just the patterns that matter, explained with humour and precision.

- Whether you are preparing for the interview loop or designing your next production system, **keep this book on your desk**. You will reach for it more often than you expect.